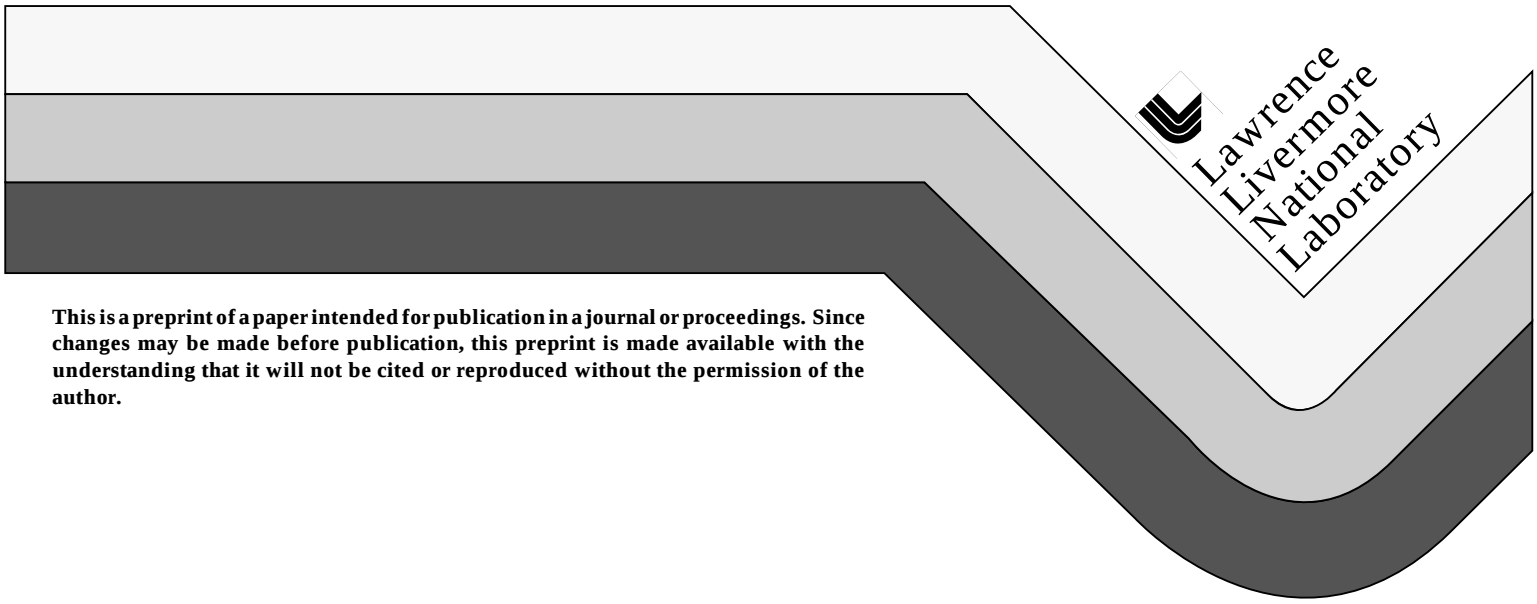
UCRL-JC-120490
PREPRINT

Using Distributed OLTP Technology in a High Performance Storage System

D.S. Fisher
T.W. Tyler

This paper was prepared for submittal to the
Fourteenth IEEE Symposium on Mass Storage Systems
Monterey, CA
September 11-14, 1995

March 1995



Using Distributed OLTP Technology in a High Performance Storage System

Terrill W. Tyler
IBM Government Systems
Houston, Texas

David S. Fisher
Lawrence Livermore National Laboratory
Livermore, California

Abstract

The design of scaleable mass storage systems requires various system components to be distributed across multiple processors. Most of these processes maintain persistent database-type information (i.e., metadata) on the resources they are responsible for managing (e.g., bitfiles, bitfile segments, physical volumes, virtual volumes, cartridges, etc.). These processes all participate in fulfilling end-user requests and updating metadata information.

A number of challenges arise when distributed processes attempt to maintain separate metadata resources with production-level integrity and consistency. For example, when requests fail, metadata changes made by the various processes must be aborted or rolled back. When requests are successful, all metadata changes must be committed together. If all metadata changes cannot be committed together for some reason, then all metadata changes must be rolled back to the previous consistent state. Lack of metadata consistency obviously jeopardizes storage system integrity.

Distributed on-line transaction processing (OLTP) technology can be applied to distributed mass storage systems as the mechanism for managing the consistency of distributed metadata. OLTP concepts are familiar to many industries such as banking and financial services but are less well known and understood in others such as scientific and technical computing. However, as mass storage systems and other products are designed using distributed processing and data-management strategies for performance, scalability, and/or availability reasons, distributed OLTP technology can be applied to solve the inherent challenges raised by such environments.

This paper briefly discusses the general benefits in using distributed transaction processing products. Design and implementation experiences using the Encina OLTP product from Transarc in the High Performance Storage System are presented in more detail as a case study for how this technology can be applied to mass storage systems designed for distributed environments.

Introduction

For several decades, on-line transaction processing (OLTP) has been a key element of commercial computing. This technology has enabled applications in industries such as banking, financial services, and reservation systems to operate properly when many users attempt to manipulate shared data. This technology enables access to and the manipulation of shared data among thousands of concurrent users. [DIE93]

OLTP systems provide mechanisms that ensure related, yet physically or logically separate, data operations either all complete successfully together as a single unit or none complete at all. We define a *committed transaction* as one in which all data operations associated with the transaction were successful and are guaranteed to be permanent. An *aborted transaction* is one in which any partial changes made associated with the transaction are rolled back to their original state.

The classic example of the utility of basic transactional concepts is a fund transfer between two bank accounts. The fund transfer transaction involves two separate data operations, the debit of one account and the credit of another. Either both operations must succeed together or neither must succeed. If, for example, the debit operation was successful but a hardware failure prevented the credit operation, the debit operation must be rolled back such that the bank account maintains its original balance.

The need for robust distributed OLTP services is increasingly being driven by the rise of distributed corporate computing environments that augment centralized legacy systems with distributed computing power [CER93] as well as the need to coordinate operations between distributed databases. Use of distributed OLTP is being driven by the creation of new distributed-system architectures.

ACID Transactional Properties

Transactional systems possess *ACID* properties: atomicity, consistency, isolation, and durability [GRA93]. *Atomicity* describes the property that all operations which

take place under a transaction form a single indivisible unit of work. The transaction either succeeds as a single unit or fails as a single unit [ORF94]. *Consistency* is the characteristic that all data operations within the same transaction either commit together or they are aborted. Aborted data operations cause modified data to be restored to its original state. The *isolation* property states that the data operations associated with separate transactions do not interfere with one another. Thus, data modifications made by one transaction which have not yet been committed are not visible to other transactions. Data updates are therefore not visible to other transactions until the data modification has been guaranteed to be permanent. This guarantee that committed data is permanently stored is referred to as the property of *durability*. Even in the event of a typical system failure at an inopportune time (i.e., before committed data is actually written to the database), once data is committed, it is guaranteed to be permanent (e.g., it will be applied when the database is restarted).

Need for ACID properties in Distributed Mass Storage Systems

A distributed mass storage system is another example of an application that requires ACID properties in managing shared persistent information. While the implementation of a distributed mass storage system does not blaze new trails in the application of OLTP concepts, it does introduce certain distinguished traits in contrast to existing commercial OLTP applications.

Distributed mass storage systems must maintain persistent distributed database information, referred to in this paper as *metadata*, describing the files it manages as well as for lower-level storage elements such as cartridges, disks, storage segments, physical volumes, virtual volumes, file segments, etc. Regardless of whether this metadata is physically distributed among numerous nodes or is centralized, the fact that multiple processes and/or threads are independently updating and contending for this metadata on behalf of multiple, concurrent end-user requests dictate that ACID properties be provided.

An even more basic need for these ACID properties can be seen when analyzing a single end-user request. For example, consider an application that creates a new file, writes a single block of data, and closes the file. A generic distributed mass storage system modeled after the IEEE Reference Model for Open Storage Systems Interconnection, would likely involve the following types of general metadata operations:

- Client application communicates with the Name Server to open the file. The Name Server creates (or reserves) a new filename metadata entry.
- Client application communicates with the Bitfile Server to continue the process of creating the file. The Bitfile Server reads various metadata and creates and/or updates metadata entries representing the new file.
- Client application communications with the Bitfile Server to write a block of data. The Bitfile Server communicates with the Storage Server, which in turn reads, creates, and updates various metadata entries. The Bitfile Server again may create and/or update metadata based on Storage Server responses.
- The Storage Server communicates with the Physical Volume Library (PVL) to request the mounting of a physical volume. The PVL communicates with a Physical Volume Repository (PVR) in order to cause a robotic device to mount the corresponding cartridge(s). Both the PVL and PVR update metadata as part of these operations.
- Client application communicates with both the Name Server and Bitfile Server in order to close the file. Both servers update metadata.

The first two operations associated with creating the file involve the Name Server and Bitfile Server. To maintain the integrity of the mass storage system database, these operations should be transactional — either both succeed together or neither take place. A situation in which the Name Server's file creation operation succeeds, but the Bitfile Server's file creation operation fails without undoing the Name Server's operation should not be permitted to happen. This would cause file entries to build up in the Name Server which had no corresponding file entry in the Bitfile Server.

This is just one example from the above scenario where multiple metadata operations between different servers/processes must be treated as a single, aggregate, transactional operation. If ACID properties are not enabled in the system, individual servers may be able to keep their respective databases correct, but, as a whole, the system metadata eventually will become inconsistent.

Use of Commercially Available OLTP Products

Developing the underlying services that provide ACID properties for metadata updates, customized for a distributed mass storage system, would be costly. Not only must the initial development of this software be considered but the additional maintenance costs throughout the product's full life-cycle must be factored in as well. Significant cost avoidance can be realized by building storage systems around commercially available OLTP products. It is the assertion of this paper that these commercial off-the-shelf products can be used successfully in these applications.

Case study — The High Performance Storage System

The remainder of this paper presents a case study in the design and implementation of a mass storage system employing an OLTP. The mass storage system discussed is HPSS, the High Performance Storage System [COY93], which uses Encina, a distributed OLTP system provided by Transarc Corporation [TRA92].

Background

HPSS is a major development project within the National Storage Laboratory (NSL). The primary development partners for HPSS are Lawrence Livermore, Los Alamos, Oak Ridge and Sandia National Laboratories and IBM Government Systems. Other partners include Cornell, NASA Lewis, and NASA Langley Research Centers.

HPSS provides a scaleable, parallel, high performance hierarchical storage system for highly parallel computers as well as traditional supercomputers and workstation clusters. A key architectural requirement is the scalability of data transfer rates, storage capacity, and the number and size of file objects. HPSS is a general purpose storage system that has been developed to scale for order of magnitude performance improvements.

To meet the high-end storage system and data management requirements, HPSS is designed to use both network-connected and directly connected storage devices and to employ parallel I/O techniques, including software striping, to achieve high transfer rates. The design is based on the IEEE Reference Model for Open Storage Systems Interconnection (Project 1244) [SSS94].

Encina provides the distributed OLTP services used by HPSS as described below. The OSF Distributed Computing Environment (DCE) is employed by Encina, and to a lesser extent directly by the HPSS servers, to provide supporting distributed computing services. Of the wide variety of services offered by DCE, HPSS and Encina make use of server multithreading, remote procedure calls (RPC), server interface registration and identification, global unique object naming, client/server authentication and authorization and other security services.

Use of Encina in HPSS

The Encina distributed OLTP system is used in HPSS to provide several essential support services:

- Distributed transaction services across multiple servers
- Nested transactions across and within servers
- A transactional record-oriented file system (the Structured File Server) for the storage of system metadata
- Transaction call-backs

Distributed transaction services across multiple servers

Encina provides, as its basic service, a distributed OLTP system which is used by HPSS servers to coordinate changes to metadata across multiple, independently-executing, multithreaded servers. Each server provides one or more application programming interfaces (APIs) defining functions that may be invoked through remote procedure calls (RPCs) from the client to the server. Many of these functions are transactional, meaning that they must be performed in the scope of a transaction created by the client before it invokes the function. Programs that initiate transactions in HPSS include the HPSS user API library and several of the servers. Encina extends and coordinates transactions across remote procedure calls (RPCs) so that the server boundaries become transparent with respect to the transactions.

In these transactional remote procedure calls (TRPCs), changes that are made to HPSS metadata as the procedure executes take on transactional characteristics. Records are locked prior to modification so that they may not be accessed or modified by other transactions until the locking transaction completes (*isolation* property). Log files are used to record the state of modified records before and after modification (*durability* property). If the transaction commits, the metadata changes made by the procedure become permanent. If the transaction aborts, the metadata changes are reversed as if they never occurred. The *consistency* property of transactions guarantees that all of the metadata changes made in the transaction either commit together or abort together. While the transaction is in progress, the new values of changed metadata records are not available outside of the transaction. This property of transactions, the *isolation* property, allows HPSS servers to make changes to system metadata records inside a transaction family without having to protect the changes from examination by other threads in the servers.

Using Encina allows HPSS to be partitioned into logically defined servers and allows the extension of transactional semantics to metadata changes that take place while performing functions in one or more of these servers. Functions such as file creation, file cataloging, file reading and writing, and file deletion typically cause a transaction to be extended across two or three servers. Transaction boundaries are chosen so that all of the metadata changes associated with one of these functions are contained in a single transaction, regardless of the number of servers involved and the number of metadata records changed. When the transaction commits, all of the records appear to change simultaneously (*atomicity* property). If the transaction aborts all of the original values of the records appear to be restored at once.

Transactions protect the integrity of system metadata during a failure of the system. At the time of failure, each transaction extant will have either committed or not. There is no “half-way” point in committing a transaction. If the transaction commits and the system then

immediately fails, the committed metadata changes will be restored during the process of restarting the system (*durability* property). If the transaction has not committed at the time of failure, it is aborted during system restart and all changes made to metadata are reversed. In either case, the integrity of the system metadata is ensured.

Encina provides a C language interface to the transaction system, Tran-C, which is used extensively in HPSS servers. Tran-C, which is implemented as a family of C macros and functions, provides easily used language constructs for creating, defining and controlling transactions [IBM94]. HPSS also makes use of a number of Encina library functions to set up transaction call-backs and control certain transaction details. Using Tran-C and the TRPC mechanism, a thread of control can create a transaction and pass it to one or more remote servers as if the remote servers were unified in a single process.

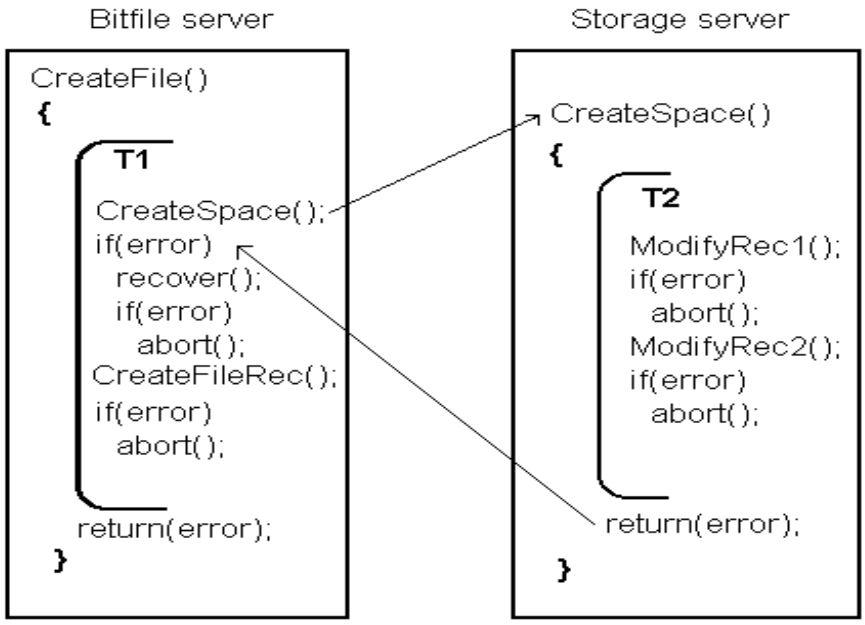
Nested transactions in HPSS

Encina provides a nested transaction facility that is used extensively in HPSS servers. In any given thread of control (which via TRPC may extend across servers) the first invocation of a Tran-C *transaction* statement causes a "top-level" transaction to be created. This transaction becomes the "top ancestor" transaction of a possible family of nested transactions. Any subsequent *transaction* statement nested in the top ancestor transaction, either in the server that created the top ancestor, or in a called server, creates a sub-transaction which becomes part of the transaction family. Sub-transactions, in turn, may create their own sub-transactions. If a sub-transaction aborts, all of its descendant transactions are also aborted, but not its ancestors. When a sub-transaction commits, any metadata changes it makes become permanent if and when all of its ancestors, up to and including the top ancestor transaction, commit.

In HPSS we isolate groups of system metadata changes from one another by creating sub-transactions in the server procedures that make the changes. If an error occurs while making a metadata change, the sub-transaction is aborted, reversing the change, and an error is returned for the procedure outcome. Other metadata changes made in other procedures as part of the HPSS operation being performed are not affected. The decision to abort these changes can be made by the caller of the failing procedure.

Without sub-transactions, an abort after an error in a called server procedure would abort the caller's transaction. All metadata changes accumulated in the transaction would be lost. This is undesirable because the caller may be able to recover from the called procedure's error by calling a different procedure or server, or taking some other corrective action. However, if the called procedure creates a sub-transaction in which it performs its metadata changes, it can abort the sub-transaction restoring the metadata to its original state and return an error code to the caller via the TRPC. The metadata changes made in the called procedure abort but the changes made in the client remain intact. The client may then recover from the error or may abort its own operation as required.

As an example of the use of sub-transactions in HPSS, consider the case of writing an HPSS file (Figure 1). The bitfile server creates a top ancestor transaction and calls a storage server, via a TRPC, to create storage space for the file. The storage server allocates space and changes its metadata in a sub-transaction. When the TRPC returns



to the bitfile server, the bitfile server links the new storage space into the file in the top ancestor transaction.

If an error occurs in the storage server, it aborts its sub-transaction, reversing its metadata changes, if any, and returns an error code to the bitfile server. The bitfile server may then either recover from the error by allocating space in another storage server, or may abort its own metadata changes, and any committed sub-transactions, and return an error to the caller of the file create operation. In each of these outcomes, the integrity of the HPSS metadata database is maintained.

Structured File Server

HPSS stores its system metadata, the information needed by each server to describe the storage objects it provides, in SFS, the Encina Structured File Server. SFS provides a record-oriented storage facility for managing large numbers of records and accessing those records using one or more keys. Access to information stored in SFS records can be either transactional or non-transactional. Transactional access exhibits the ACID properties described earlier.

SFS can associate locks with the records in its files and binds these locks to transactions. The locks segregate access to the records to transactions, providing the *isolation* property of transactional access to HPSS metadata files. SFS records changes made to records in transactions in a log which is used to record committed changes.

Each transactional HPSS server maintains one or more system metadata files in SFS. Records in HPSS metadata files are indexed using a primary key and, in many cases, one or more secondary keys. In general, primary keys are unique within HPSS metadata files, whereas secondary keys may or may not be unique.

The objects defined by HPSS servers are usually connected to one another within a server, and across servers, via primary and secondary keys embedded in the metadata records. Keys are placed in records so that the HPSS database taken as a whole may be traversed from either the top down or the bottom up. The system metadata architecture is optimized for top-down traversal. For example, each storage segment record maintained by a storage server is keyed by a storage segment key. The record has embedded in it the key of the virtual volume from which the storage segment is allocated. This key is a secondary key for the storage segment record, while the storage segment key is the primary key. Several storage segments may be allocated in the same virtual volume, so the secondary virtual volume keys will be shared by several storage segment records and will not be unique in the storage segment metadata file. Given a storage segment key, the associated virtual volume can be located immediately by searching the storage segment metadata file with the unique key. Only one record will be returned

and it will contain the virtual volume key. This is top-down traversal of the database.

Given a virtual volume key, the storage segments located on the volume can be found by indexing the storage segment metadata file by the virtual volume secondary key. Zero or more records will be returned, each a storage segment allocated in the virtual volume. This is bottom-up traversal of the database.

In a similar fashion, bitfile metadata records are linked to storage segments (top-down), and storage segment are linked to bitfiles (bottom-up). Name server entries are forward-linked to bitfiles and bitfiles are reverse-linked to name server entries. Top-down traversal of the database is used to locate file metadata information to carry out ordinary client requests (create / read / write / delete). Bottom-up traversal of the database is used to carry out certain maintenance activities, for example, locating all of the files stored entirely or partially on a failed physical volume.

Transaction call-backs

Encina provides a facility in which a process may arrange for a procedure to be called when a selected event occurs during the processing of a transaction. These events include transaction preparation, transaction resolution, transaction abort, and others, and can be established for top-level or nested transactions. The procedure call-back is made by an Encina library routine, in a thread managed by Encina, when the requested event occurs. From the point of the view of the process, the procedure call happens asynchronously.

HPSS uses the call-back mechanism extensively to coordinate transactions with various maintenance activities in the servers. For example, several servers use call-backs to maintain a pool of open file descriptors (OFDs) for the SFS metadata files. An OFD is a file access handle provided by SFS. When an OFD for a metadata file is needed by the server, it is taken from a free pool and used in a transaction, which causes the OFD to be associated with the transaction. A call-back is set up by an HPSS library routine when the OFD is taken from the pool to call a server procedure that returns the OFD to the free pool. In this case, the call-back occurs when the top ancestor transaction resolves (either commits or aborts).

In the HPSS disk storage server, call-backs are used to implement an in-memory cache of active metadata records. In many cases, this cache allows the server to reference metadata without taking the time to read it from the metadata file. Records are locked when they are referenced using the identifier of the top ancestor transaction as the key. The cache entry remains locked to the transaction until the transaction resolves. Sub-transactions may refer to the cache entry once it is locked by using the same top ancestor transaction ID. Other transactions and sub-transactions are blocked from using the entry until the locking transaction resolves.

Cache lock management and consistency is maintained through transaction call-backs. A call-back is set up when the cache entry is first locked by a transaction, to unlock the cache entry and make it consistent with the metadata file. The call-back occurs after the top ancestor transaction finishes, when the resolution of the transaction (commit or abort) is known. The disposition of the cache entry depends on the operation that was done and the outcome of the transaction. If the entry was simply read, it is unlocked. If the entry was created during the operation and the transaction committed, the entry is left in the cache. If the transaction aborted, the entry is deleted from the cache. If the cache entry was deleted during the operation and the transaction was aborted, it is restored to the cache (and refreshed from the metadata file). If the transaction committed, the entry is permanently deleted from the cache.

If the cache entry is modified during the transaction, the entry remains in the cache but is refreshed from the metadata file each time it is referenced until the transaction resolves. This is done because tracking the outcomes of sub-transactions as they change the entry becomes too difficult and would require that we reproduce a significant part of the SFS transaction management algorithm. If a sequence of modifications are made to a cache entry we simply refresh the cache entry before each modification and let SFS handle the

Operation	Transaction Commit	Transaction Abort
Read	Unlocked	Unlocked
Write	Refreshed	Refreshed
Create	Unlocked	Deleted
Delete	Deleted	Refreshed

Figure 2: Actions taken on metadata cache objects by operation and transaction outcome.

complexities of keeping track of which sub-transactions have committed and which have aborted. Once the top ancestor resolves, the call-back routine refreshes the cache entry once more from the metadata file and then unlocks it. Voiding the cache entry during metadata updates does not represent a serious degradation of the cache performance because most top-level transactions perform only one modification to a cache entry before committing, so the cache entry is usually refreshed only once, when the transaction finishes.

Implementation Issues

The balance of this paper describes Encina related issues that arose during the design and implementation of HPSS. These issues include:

- Interaction of transactions and storage media
- Dealing with long duration transactions
- Choosing transaction boundaries
- Dealing with transaction side-effects

Interaction of transactions and storage media

End user read and write operations made to HPSS files are done transactionally. Changes made to the HPSS metadata records for a file being read or written are made within a transaction created by the bitfile server so that they may be reversed in case of certain rare errors. An error writing a bitfile server metadata file is an example of the sort of error that will abort a read or write operation. In this case, if the associated metadata changes made by the supporting storage server are not also reversed, the HPSS database would quickly become inconsistent. Note that storage media errors, including unexpected End-Of-Media errors do not cause the I/O operation to abort. An error is returned for the operation, but the appropriate metadata changes are committed.

Many storage media have the property that they may be positioned, written, repositioned to some point that was previously written, and written again. Other media may only be written from the point where writing last ended. When writing user data, HPSS retains a record of the starting point, a tape or disk address, in system metadata that was committed some time in the past. At the end of the write operation, the updated Next-Address-To-Write is written into the system metadata in a transaction. If the transaction later commits, the Next-Address-To-Write information becomes permanent and the user data becomes committed to the system.

However, if the transaction is subsequently aborted, for some reason noted above, the previous Next-Address-To-Write is restored to the system metadata and the user data is forgotten. If the storage media is repositionable, the next write operation to the media will simply start in the same place as the aborted operation and overwrite the aborted user data. Thus, the transactional semantics of the write operation lead to the automatic recovery of storage media that might otherwise have been wasted.

If the storage media is not repositionable, an attempt to write the media at the restored starting point will fail because the write operation that aborted earlier moved the actual Next-Address-To-Write forward on the media, then forgot it. In this case, the HPSS metadata and the storage media are not synchronized with one another because the storage media cannot reverse a write operation.

To resolve this situation, HPSS will recover from the write error by moving the write operation to a different volume and retrying, and by marking the volume that is out of step with the system metadata as temporarily un-writable. Later, a system utility will find this volume, locate its actual next write address and update the HPSS system metadata accordingly. The steps of marking the volume un-writable and correcting the volume's Next-Address-To-Write are done in top-level transactions, isolated from any other transaction that may be in effect.

Note that this "transactions vs. storage media" problem is not a serious one for HPSS because of two factors:

- Aborts in user data write operations are rare - metadata changes will usually commit even if the write operation ends in error
- Storage media maintenance activities such as tape repacking and reclaiming are always going on

Dealing with long duration transactions

In order to make transactional changes to the HPSS metadata, the HPSS servers use SFS to create locks on records that are being changed. The locks implement the isolation property of Encina transactions and are maintained on a per metadata record basis by SFS. If a transaction locks a metadata record with the intent of writing the record (write-lock), all other transactions are blocked from reading or writing the record. They may not read the record unless they are willing to read the current committed data (reading without a lock).

In many typical OLTP applications transactions are of short duration. In a mass storage system some transactions may be very short while others may be very long in duration. Short transactions include operations

that simply perform changes to the system metadata (e.g. set attributes functions) while long duration transactions are those that perform I/O to the archive.

Holding locks on HPSS metadata for long periods of time while long duration I/O transactions are in effect, creates two potential problems for HPSS. The first is blocking other transactions' access to metadata records. While the isolation property of transactions is vital to maintaining the consistency of the HPSS database, locks held on metadata records by long duration I/O transactions could cause other transactions to block for long periods of time while trying to perform non I/O operations such as allocating storage space.

HPSS resolves this problem by imposing restrictions in the way storage objects are used and reserving objects to active I/O operations. For example, files must be opened through the bitfile server prior to requesting a read or write operation. When I/O begins on the file, the file is reserved to the client until the I/O completes. If another client attempts an I/O operation, it is blocked until the first client's operation finishes and the transactions created during the I/O operation have resolved.

In the storage server, the *session* construct is used to group I/O operations together and reserve system resources. The bitfile server opens a session with the storage server each time it performs a file open operation. Initially, the session has no system resources assigned to it. When the first I/O operation on the file is processed by the bitfile server, the associated storage server assigns the corresponding storage segments, virtual volumes and physical volumes to the session, if they are not already assigned to other sessions. Locks can then be taken on these objects during the I/O operation without a blocking hazard because the objects have first been reserved to the session. If any of the resources needed by an I/O operation are already reserved in another session, the I/O operation is blocked until the session holding the resources completes. This prevents I/O transactions from blocking on metadata locks and prevents deadlocks that would occur if two or more transactions competed for shared metadata records.

The second problem in dealing with long duration I/O transactions is the issue of dealing with failed transactions after server crashes, host system failures, power failures, etc. For reasons such as these, a transaction will occasionally fail to complete. The Structured File Server, which acts as the transaction coordinator in HPSS, maintains the integrity of the HPSS database by aborting the transaction. This is done by recording a time of last use in each of the Open File Descriptors open on HPSS metadata files. If the time since an OFD was last used exceeds a configurable limit set in SFS (the SFS OFD idle time-out), and another transaction attempts to take a lock on a record locked by the expired OFD, the expired transaction is aborted and the expired OFD is closed. This feature protects the HPSS database from the failures noted above and is an important reason for using an OLTP in a mass storage system.

If the transactions in the HPSS database could be characterized as not exceeding some upper bound in duration, the SFS idle time-out value could be set to that upper bound. However, when doing I/O operations on files of arbitrary length, as HPSS is designed to do, an upper bound cannot be established. Small values of the SFS idle time-out would effectively limit the size of files that could be written into HPSS. Large values would have an undesirable effect on recovering HPSS after a failure. One would either have to wait a long time for the SFS idle time-outs to expire on the transactions in effect at the time of the failure, or re-initialize SFS which increases the time to re-initialize the system. Furthermore, for whatever practical value the SFS idle time-out is set to, an I/O operation that takes longer can be imagined. Note that time spent waiting for storage volumes to be mounted in a busy system may be spent inside the context of a transaction, further increasing the likelihood of an aborted transaction.

HPSS resolves this situation by employing a technique suggested by developers at Transarc. When OFDs are assigned to a transaction a background thread in the server periodically performs a benign SFS operation using the OFD. These operations take very little time and don't change the state of locks on metadata records, but do update the time of last reference in the OFD. The period of this "keep-alive" operation is set to a value that is most of the OFD idle time-out, but allows a margin of safety. As long as the server is functioning, its transactions and OFDs stay valid and I/O operations can take as much time as they need to complete. Locks on system metadata remain valid, competing accesses to the locked records remain blocked, and the long duration I/O operations will complete normally.

Choosing transaction boundaries

In a few instances during the implementation of HPSS we located the boundaries of transactions in places different from those described in the system design. Top-level transactions were used in some places that were originally implemented as sub-transactions. In other cases, operations that seem logically to be a single transaction were implemented in two transactions.

A good example of this appears in the bitfile server. There are two steps in the loop that processes user write functions. In the first step storage space is allocated; in the second step the space is written with the user data. However, there are a few rare errors that can occur in the second step that make it desirable to discard the space allocated in the first step. The two steps logically form a single transaction, but in the implementation of the function, two transactions are used. If the function could be performed in one transaction, space allocated in the first step could be deallocated by simply aborting the transaction.

However, since the second step is sometimes a long duration operation, a single transaction that includes the storage allocation step will hold a storage map locked during the entire write operation. In the HPSS tape storage server this is a tolerable situation because storage maps are kept in a busy state during tape writes. When a tape is being written, no other storage space can be allocated on the tape until the write operation completes.

In the HPSS disk storage server, however, holding the storage map busy during a long disk write is undesirable because storage space for disk segments is allocated in finite sized blocks, rather than in an open-ended fashion as for tape. Disk storage maps are therefore not kept in a busy state while the segment is being written. We want to allocate the disk space, modify the disk storage map, and commit those modifications as soon as possible so that other storage segments may be allocated from the map.

To resolve this problem, the HPSS bitfile server allocates storage segments in top-level a transaction, writes a log with the segment identification information, and commits the transaction. Then it starts a second top-level transaction, writes the user data, links the storage segment to the file metadata, and deletes the log entry. In the same transaction, the storage server updates its metadata with information about the written length of the segment. The first transaction creates the storage segment and releases the storage map, the second writes the user data, updates the storage segment metadata and attaches the storage segment to the file metadata. If an error occurs during the file write operation, we delete the storage segment in a separate transaction. We use two transactions where we originally expected to use one so that locks on metadata objects are not held for long periods of time, blocking the use of the objects by other users.

In another case, a deadlock that occurred when two or more HPSS clients were reading files located on the same virtual volume was broken by changing a sub-transaction into top-level transaction.

To carry out a file read operation, the bitfile server creates a top-level transaction in which the file statistics changes are made. When two clients accessed two files, located on a shared virtual volume, a deadlock occurred. One client would mount and read one volume, and in so doing would cause sub-transactions to be started that made minor metadata changes associated with mounting the volume (time last mounted, number of mounts). The other client would do likewise with a second volume. Then each client would attempt to mount and read the volume previously read by the other. Since each client is holding a lock on the volume metadata needed by the other, in an uncommitted sub-transaction, neither could proceed.

One solution to this deadlock is not to make the metadata changes associated with mounting the volumes. Another is to make the changes in a top-level transaction. We chose the second route recognizing that it is appropriate to record the metadata changes associated with mounting the volume regardless of the outcome of the read operation. The physical act of mounting the volume cannot be undone-done therefore the records should be kept. The volume mount statistics are updated in a top-level transaction embedded in the bitfile server's data reading transaction.

Dealing with transaction side-effects

In a distributed OLTP system, transactions take a certain amount of time to become completely finished after they commit In SFS, locks on records may be held for a short time until the server finishes the transaction. During this period of time searching a system metadata file with a non-unique secondary key can yield unexpected results.

In the HPSS storage server, we index storage maps by a primary key, which is unique for each map, and a secondary key, which may be shared by many maps. The state of the map (free, busy, full, etc.) is part of the secondary key. When the server creates a storage segment, it searches the map metadata file by secondary key, searching for free maps.

While testing the segment creation logic in the tape storage server, we discovered that a rapid serial sequence of storage segment creation requests were resolved on a series of volumes, not on a single volume as expected. If only one volume was available in the system, the sequence failed with a "no space" error when in fact space was available.

The key to understanding this effect is to recognize that distributed transactions take time to resolve. When the storage server changed a storage map's state from busy to free in one transaction, then searched the metadata file for free maps in a new transaction very shortly thereafter, it failed to find the map modified in the first transaction. It then either allocated space on a different volume, or if no other volume was available, returned the "no space" error.

This situation is resolved in HPSS in two ways. First, the problem resolves itself if requests to create tape storage segments are separated by enough time for the transactions to finish in SFS. Second, if a storage server client wishes to create a string of segments on the same volume, an option is provided in the segment creation function that causes the storage server to read the desired storage map directly, by primary key, rather than indirectly searching for a free volume by secondary key. The direct read operation blocks until any transactional activity on the desired record is resolved. The delay is minimal and the system maintains transactional integrity in its metadata changes.

Conclusions

Commercial OLTP technology can provide the ACID properties required for new distributed mass storage systems in managing distributed metadata information. HPSS is an example of a new storage system designed to

take advantage of an existing OLTP product. In doing so, development and maintenance costs have been greatly reduced while overall system reliability has been enhanced.

We have shown examples of the use of OLTP systems in the implementation of a high performance mass storage system, and shown how those algorithms can be modified to meet the requirements of both the mass storage system and the OLTP system.

Distributed mass storage systems present unique issues and challenges in the application of OLTP technology. However, given sufficient capability from the OLTP product, these issues and challenges can be successfully managed and overcome.

This work was performed, in part, by the Lawrence Livermore National Laboratory under the auspices of the U.S. Department of Energy under contract No. W-7405-Eng-48, Los Alamos National Laboratory, Oak Ridge National Laboratory and Sandia National Laboratories, under auspices of the U.S. Department of Energy Cooperative Research and Development Agreements, by Cornell, Lewis Research Center and Langley Research Center under auspices of the National Aeronautics and Space Administration and by IBM Government Systems under Independent Research and Development and other internal funding.

References

[COY93] Coyne, R. A., H. Hulen and R. W. Watson, “The High Performance Storage System”, Proc. Supercomputing 93, Portland OR, IEEE Computer Society Press, Nov 1993.

[SSS94] IEEE Storage Systems Standards Working Group (SSSWG) (Project 1244), “Reference Model for Open Storage Systems Interconnection, Mass Storage Reference Model Version 5”, Sept 1994. Available from the IEEE SSSWG Technical Editor, Richard Garrison, Martin Marietta (215) 532-6746.

[GRA93] Jim Gray and Adreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[DIE93] Scott Dietzen and Alfred Spector, Distributed Transaction Systems, *Distributed Computing Environments*, McGraw-Hill, pp. 223-257, 1993.

[CER93] Dan Cerutti, The Rise of Distributed Computing, *Distributed Computing Environments*, McGraw-Hill, pp. 3-8, 1993.

[ORF94] Robert Orfali, Dan Harkey, Jeri Edwards, *The Essential Client/Server Survival Guide*, Van Nostrand Reinhold, p. 242, 1994.

[TRA92] Transarc Corporation, *Encina Product Overview and Encina Product Documentation*, 1992.

[IBM94] IBM Corporation, *Encina Transactional-C Programmer's Guide and Reference for AIX*, SC23-2465-02, 1994.

Technical Information Department • **Lawrence Livermore National Laboratory**
University of California • Livermore, California 94551